SOFTWARE ENGINEERING LABORATORY SERIES

SEL-81-305SP1

Ada Developers' Supplement to the Recommended Approach

NOVEMBER 1993

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Ada Developers' Supplement to the *Recommended Approach*

OCTOBER 1993

FOREWORD

The **Software Engineering Laboratory** (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effects of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

The primary authors of this document are **Rush Kester** and **Linda Landis** of Computer Sciences Corporation.

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

ACKNOWLEDGMENTS

The recommendations that appear in this document were formulated by the Ada Process Working Group (APWG). The members of the APWG — experienced Ada developers from NASA/GSFC and CSC — met regularly from October 1992 to January 1993 to discuss and resolve issues of concern to Ada projects in the flight dynamics environment of NASA/GSFC. The group reached consensus on a number of guidelines, which have been collected into the present volume.

Ada Process Working Group

Eric Booth, CSC
Jerry Doland, CSC
Gerry Heller, CSC
Ed Joyce, CSC
Rush Kester, CSC
Warren Miller, CSC
Wendy Shoan, GSFC
Michael Stark, GSFC
Warren Steger, CSC

ABSTRACT

This document is a collection of guidelines for programmers and managers who are responsible for the development of flight dynamics applications in Ada. It is intended to be used in conjunction with the *Recommended Approach to Software Development* (SEL-81-305), which describes the software development life cycle, its products, reviews, methods, tools, and measures. The *Ada Developers' Supplement* provides additional detail on such topics as reuse, object-oriented analysis, and object-oriented design.

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

The *Ada Developers' Supplement to the Recommended Approach* is a collection of guidelines for the development of flight dynamics applications in Ada. These guidelines represent the "best current practice" of Ada software developers working on projects monitored by the Software Engineering Laboratory (SEL). They were formulated by the members of the Ada Process Working Group (APWG), a committee composed of experienced Ada developers from the Software Engineering Branch of the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and the Software Engineering Operation of Computer Sciences Corporation (CSC).

The recommendations in this document are intended to supplement the guidance provided in the SEL's *Recommended Approach to Software Development* (Reference 1). The *Recommended Approach* describes the software development life cycle, its products, reviews, methods, tools, and measures. This *Ada Developers' Supplement* provides additional detail, primarily in the areas of reuse and object-oriented approaches, that will further assist managers and developers on Ada projects. It does not attempt to be a comprehensive treatment of either the life cycle or the methodologies of Ada software development.

The document loosely follows the order and organization of the *Recommended Approach*, although material that relates to consecutive sections in the Recommended Approach has been consolidated. Consequently, the current document contains a single section containing information applicable to both requirements definition and requirements analysis, another section for preliminary and detailed design, a section for implementation, and a final section for system and acceptance testing.

To use this *Supplement* effectively, the reader should be familiar with the *Recommended Approach*, trained in the Ada language, and conversant with object-oriented analysis/design.

SUPPLEMENT TO SECTION 2 OF
THE *RECOMMENDED APPROACH*

THE SOFTWARE DEVELOPMENT LIFE CYCLE

TAILORING THE LIFE CYCLE FOR HIGH-REUSE PROJECTS

Ada projects that reuse their software architectures and the majority of their source code should use a streamlined life cycle tailored for high-reuse systems. The key areas in which such a life cycle differs from the standard life cycle documented in the *Recommended Approach* are

- During the requirements analysis phase, the development team uses prototyping to verify that reused code meets project requirements, especially requirements for performance.

- On projects in which the system architecture is already established by the reused system, a separate preliminary design phase is unnecessary. Instead, all design activities take place during a single *design phase.*

- During the design phase, the development team holds walkthroughs with the analysis team to cover new and modified components. Walkthroughs need not cover components that will be used verbatim.

- The design of the reused system may be documented more accurately in its system description than in its design document, particularly if the design document was not updated to reflect modifications made late in the life cycle. In such a case, the development team reuses relevant portions of the existing system description to prepare a *preliminary system description* as the baseline design document for the new system. An appendix is added to this preliminary system description to provide an overview of system operations.

- The development team holds a combined preliminary design review (PDR)/critical design review (CDR) to formally review and approve the design. The team does not present the entire system design at this review. Instead, presenters focus on the differences between the design of the current system and that of the reused system.

- During unit and integration testing, developers concentrate on new and modified components.

- Test teams conduct system/acceptance testing for the entire system, just as they would if the components were newly developed.

SUPPLEMENT TO SECTIONS 3 AND 4 OF
THE *RECOMMENDED APPROACH*

THE REQUIREMENTS DEFINITION AND REQUIREMENTS ANALYSIS PHASES

METHODS AND TOOLS

The following paragraphs extend the discussion of object-oriented analysis that appears in Section 3 of the *Recommended Approach.*

Object-Oriented Analysis

The purpose of object-oriented analysis is to identify and understand the objects in the *problem domain,* i.e., the problem space that is rooted in the real world. In the flight dynamics environment, the problem domain is described in the requirements document for the system. The *solution domain* is implemented by a combination of software and hardware constituting the actual system. (The solution domain is the focus of object-oriented *design.*) Figure 1 illustrates this concept.

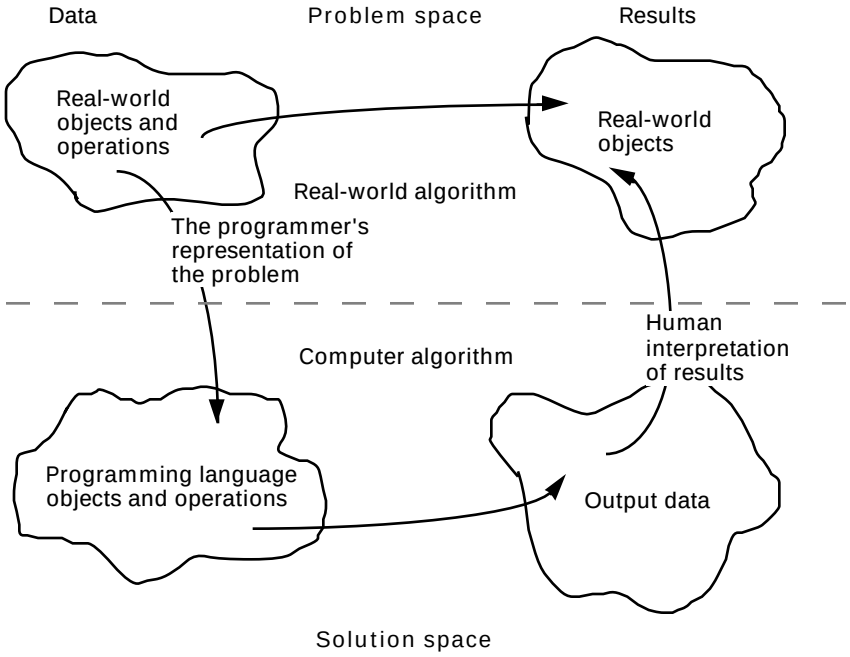


Figure 1. Problem and Solution Domains
[Adapted from Figure 5-1 of G. Booch,
Software Engineering With Ada *(Reference 2)]*

The process of object-oriented analysis consists of the following steps:

1. Identify the objects in the problem domain.

2. Identify the attributes of each object (e.g., input, output, or state information).

3. Identify the relationships between objects.

4. Map functions and operations to objects.

5. Optionally, specify an object's behavior via behavior modeling or state-transition modeling.

6. Optionally, group objects into abstract classes (e.g., sensors or environmental factors).

During the design phase(s), developers identify the objects that are needed in the software solution to the problems posed in the requirements. Steps 2 through 6 are then repeated for these solution-domain objects.

REFERENCE

For a complete treatment of object-oriented analysis, see P. Coad and E. Yourdon's, *Object-Oriented Analysis* (Reference 3).

PRODUCTS

Software Development/Management Plan (SDMP)

Additional information should be included in the SDMP of an Ada project, as follows:

- In the section entitled *Build Strategy* (Item 3.6), discuss how Ada compilation dependencies affect the build plan.

- Under *Configuration Management* (Item 5.3), address the way in which Ada libraries will be kept synchronized with source code (i.e., CMS) libraries.

SUPPLEMENT TO SECTIONS 5 AND 6 OF
THE *RECOMMENDED APPROACH*

THE PRELIMINARY DESIGN AND DETAILED DESIGN PHASES

For an Ada system with many new components, more effort is usually expended in the preliminary design phase than is spent during the preliminary design of a comparable FORTRAN system. The Ada development team holds the PDR only after the software architecture has been defined, i.e., after all packages and subprograms in the problem domain have been described. Consequently, the PDR is usually held later than it would be on a FORTRAN project, and a larger percentage of the total design work is completed prior to PDR.

METHODS AND TOOLS

The guidelines in this section primarily address the methods and tools that Ada developers use to create an object-oriented design. They include methods for determining the optimum design as well as methods for documenting the design, such as object diagrams, prologs, and program design language (PDL).

Reuse Verification

As a part of the process of reuse verification (described in Section 5 of the *Recommended Approach*), the development team must thoroughly review any software components that are slated for reuse to determine the extent to which project dependencies are parameterized. Developers examine relevant portions of the system description, the requirements and specifications, and any test drivers identified in the system/acceptance test plans. Where project dependencies *are* parameterized, the development team identifies the effort needed to supply their project's specific parameters.

If software components are being imported from a different computing environment or from outside the flight dynamics environment, the team should assess potential portability problems by attempting to compile and execute the components. Use of implementation-specific pragmas, references to missing source files, or references to machine-specific run-time library routines are easily identified when the software is compiled and linked. Missing data files and subtle differences in implementation-

defined features, such as input/output, tasking, or arithmetic precision, may not be evident until the software is executed.

If the reuse candidates don't completely satisfy the project's needs, the development team must decide which components they will

- reuse, making modifications that are specific to the current project;
- reuse, making modifications that generalize or parameterize the component(s) for future as well as current reuse; or
- design and code newly.

NOTE

During reuse verification, developers should submit questions to the requirements definition team. The requirements definition team can help to identify commonalities between the current project's requirements and those of other past, present, or future projects.

The development team must make these decisions based on the priorities of their particular project (e.g., low cost, reusability in future systems, or high run-time performance). The team must evaluate the reuse possibilities and weigh the effects of various tradeoffs in view of these project drivers. When modifications are necessary to enable or improve reuse, the development team's estimates of the effort required should be specified in the SDMP.

Where no candidates for reuse exist, the development and management teams should consider making the new components reusable on future projects. To accomplish this, the team performs a limited domain analysis, identifying common elements of the software and isolating and parameterizing elements that vary from project to project.

Prototyping

In the flight dynamics environment, prototyping is used to mitigate risks related to new technology, to resolve requirements issues, and to model system performance. During preliminary design, the development team should use prototyping to assess potential performance problems. In addition, the team may use prototypes to evaluate tradeoffs between design alternatives being considered. These prototypes allow the development team to gain experience in the environment and in the application or Ada language features, which generally results in a better system design.

In a stable computing environment and application domain (where a prior project serves as a prototype), little or no prototyping of new technology is usually necessary. However, when developers anticipate significant differences in the

environment or application, they should use prototyping to assess and/or reduce technical risks. Often these prototypes are planned in earlier phases but implemented during preliminary design.

Object-Oriented Design

Preliminary design starts with the conceptual object model developed during the requirements analysis phase. This conceptual model identified problem-domain objects and their attributes, defined operations on those objects, and established the interfaces between them. During preliminary and detailed design, the development team maps these objects to Ada units and specifies the approach to be taken in implementing them.

In the preliminary design of an Ada system, the development team identifies all packages and subprograms that address elements of the problem domain. This includes all high-level objects needed to implement the system capabilities, the functions and procedures affecting these objects, externally visible data, and all of the object's interfaces and dependencies. The development team defers identification of the lower-level components that implement solution-domain objects or utilities until the detailed design phase. However, the development team must define the overall system architecture before the preliminary design can be considered complete.

The development team documents the preliminary design in the preliminary design report, using a combination of graphical diagrams, narrative text, Ada specification code, PDL, and descriptive commentary. At the highest level, the preliminary design is depicted in a system context diagram and accompanying narrative text. The context diagram shows the external interfaces to the system and the programs that make up the system; arrows indicate the flow of data to and from the system. The context diagram is accompanied by narrative text that describes both the programs and their external interfaces. An example of a context diagram is shown in Figure 2.

The development team then refines each program into top-level subsystems. These top-level subsystems represent the highest level of objects in the system. A top-level object diagram is used to depict these objects and their interdependencies, as shown in Figure 3. Developers continue to refine problem-domain objects until all subsystems, all packages, and all visible subprograms within those subsystems have been identified.

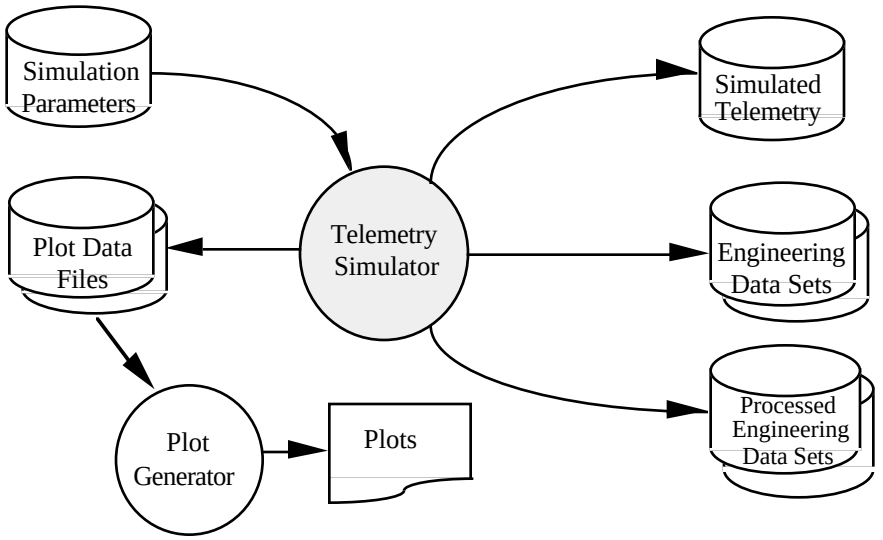


Figure 2. System Context Diagram

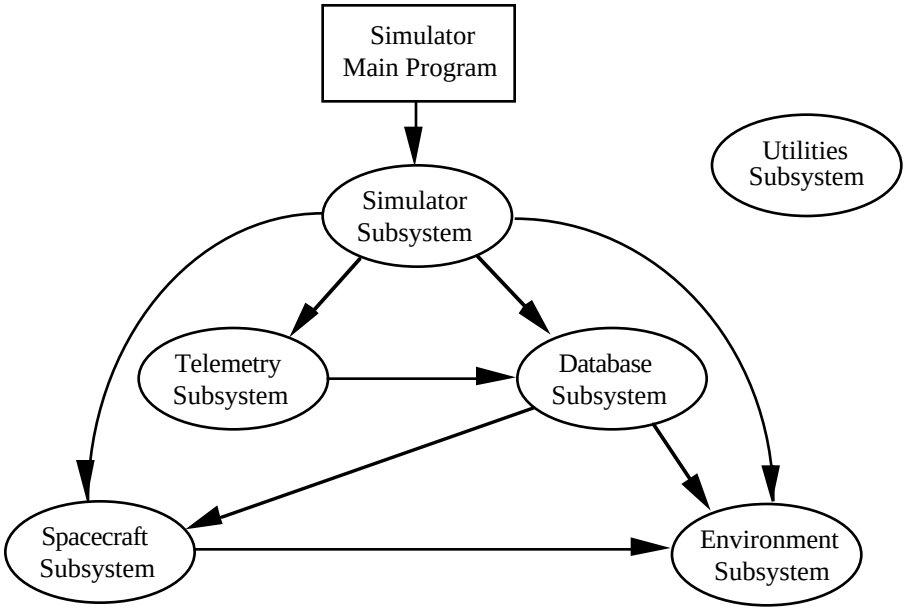


Figure 3. Top-Level Object Diagram

Guidelines for Preliminary Design

To reiterate, the primary goal of the preliminary design phase is to translate the conceptual object model that was developed during the requirements analysis phase into Ada units. Ada units can take a variety of forms: packages, procedures, functions, or their generic counterparts. Related library units can be grouped together into logical subsystems to simplify documentation. Library units can also contain nested units, i.e., regular and generic packages, procedures, and functions, as well as tasks.

The development team's selection of object groupings and Ada units will have a significant impact on the ease with which the system can be implemented, maintained, and reused. In some cases, as described in the *Ada Efficiency Guide* (Reference 4), these choices impact the performance of the system. Because of this, the development team should examine alternative mappings and review the advantages and disadvantages of each before settling on a preliminary design.

The preliminary design chosen must meet project requirements and satisfy project drivers. Developers must always keep in mind their project's unique set of priorities, even as they strive to create a design that is modular, reusable, understandable, testable, and maintainable. No hard and fast rules can be used to achieve the necessary balance, although the following rules of thumb apply. Note that the first four of these rules apply to *any* software design, whether functionally- or object-oriented.

- The higher the internal cohesion and the lower the external coupling, the better the modularity.

- The better the modularity, the easier a system is to understand, implement, test, maintain, and reuse.

- The easier a system is to understand, the more reliably it can be built and the easier it is to use.

- A software structure with strictly top-down dependencies is easier to develop, maintain, and reuse than one that has bottom-up or sibling dependencies.

- Objects are usually implemented as Ada packages, because packages allow the grouping of data types and multiple operations as a single Ada library unit.

- Use of Ada generic units can provide some of the design flexibility and reuse potential of object-oriented classes. With this approach, the generic template defines those properties common to (and inherited by) all objects in the class, and generic parameters (especially generic subprogram parameters) define those properties unique to specific objects in the class.

- Generic Ada units, parameterized to reflect variability in the application domain, are more reusable than their non-generic equivalents. However, generic Ada units are more abstract and are harder to understand than their non-generic equivalents. Generic units require additional in-line commentary and external documentation that explains how to reuse them.

Identifying Programs and Ada Tasks

Ada systems are made up of one or more executable programs. Historically, Ada systems such as telemetry or dynamics simulators have been designed as a single program. However, as larger applications are developed in the future, this may no longer be the case. Consider defining separate executable programs when one or more of the following conditions exist:

- Functions must execute on different computer processors.

- Functions execute at different times in the operational scenario.

- A checkpoint is required between functions for manual output validation, input editing or restart/recovery. (Functions before the checkpoint should be in a separate program from those that follow.)

- Some functions (or their associated data sets) require restricted access controls. (Isolate the restricted functions/data sets.)

- Concurrency is required but vendor interface routines do not support concurrent execution of Ada tasks.

Within Ada programs, one can achieve concurrently executing control flows by use of Ada tasks. Ada tasks are used for one or more of the following reasons:

- To manage multiple, asynchronous I/O devices

- To model real-world, concurrent processes

- To synchronize concurrent access to state data (Note that, in Ada 9x, protected types replace tasks as a mechanism for locking and unlocking state data.)

Methods for Documenting External Interfaces

Typically, the first objects that the development team specifies with Ada code are the interfaces to external systems. It is good practice to document these interfaces by declaring the Ada records that correspond to the interface data. When the interface mechanism is a file, describe the file's organization and access protocol. Encapsulate the record declaration and access routines in the package that implements the interface object.

When the interface is to non-Ada programs, to programs generated by a different Ada compiler, or to programs executing on a different type of computer, it is generally advisable to include Ada representation clauses to completely specify the record layout. It is up to the development team, however, to decide whether the physical representations should be specified during preliminary design or deferred to a later phase.

Object Dependency Diagrams

An object dependency diagram shows the WITHing dependencies among each program's Ada library units, i.e., the main procedure, packages, and standalone subprograms. The dependencies of the specification, body, and any subunits for a given Ada library unit are combined and the combined unit is shown as a single icon.

Each project should select a graphic notation for object diagrams based on available tools, project needs, and development team preferences, and should use the selected notation consistently throughout the life of the project. The SEL recommends the GOOD notation (Reference 6), although the notation of Grady Booch (Reference 5) is an accepted alternative, since it is supported by a number of CASE tools. The icons for these notations are shown in Figures 4 and 5.

In order to provide the best overview of a program's structure, as many units as is practical should be shown on a single page. Where not all units can be shown, units that contribute little to an understanding of the program's structure may be omitted from the diagram (although they should at least be listed). Alternatively, the diagram may be continued on a subsequent page. An example of an object dependency diagram is given in Figure 6.

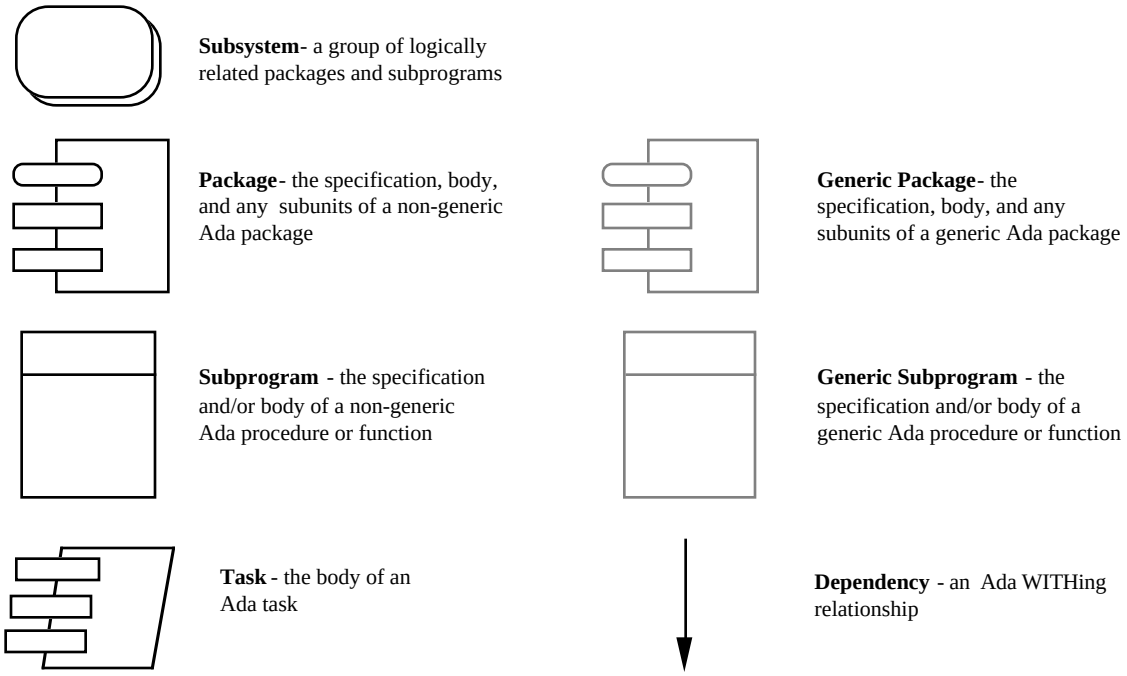


Figure 4. Graphic Icons Used in Booch's Notation

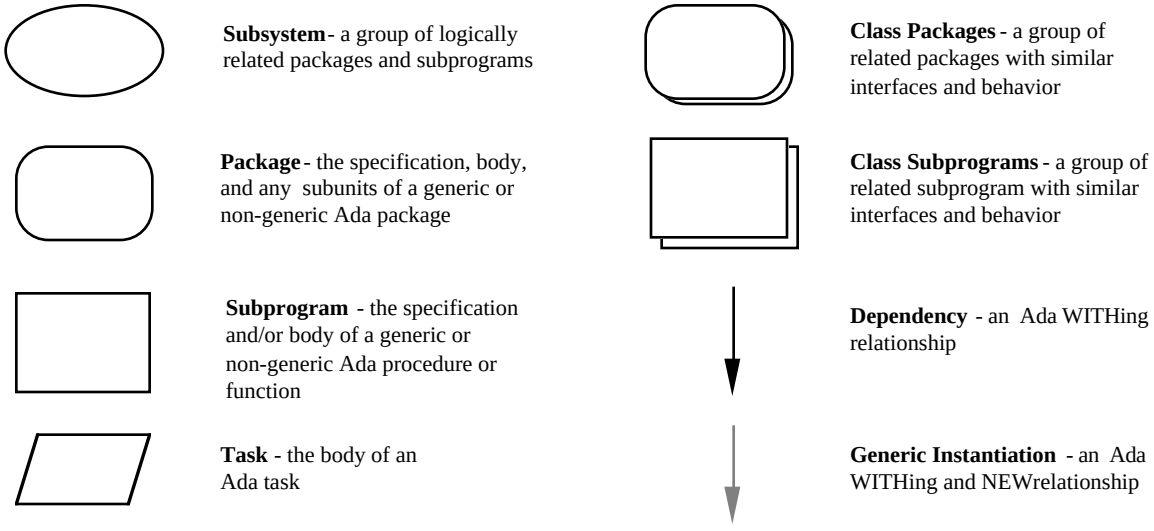


Figure 5. Graphic Icons Used in the GOOD Notation

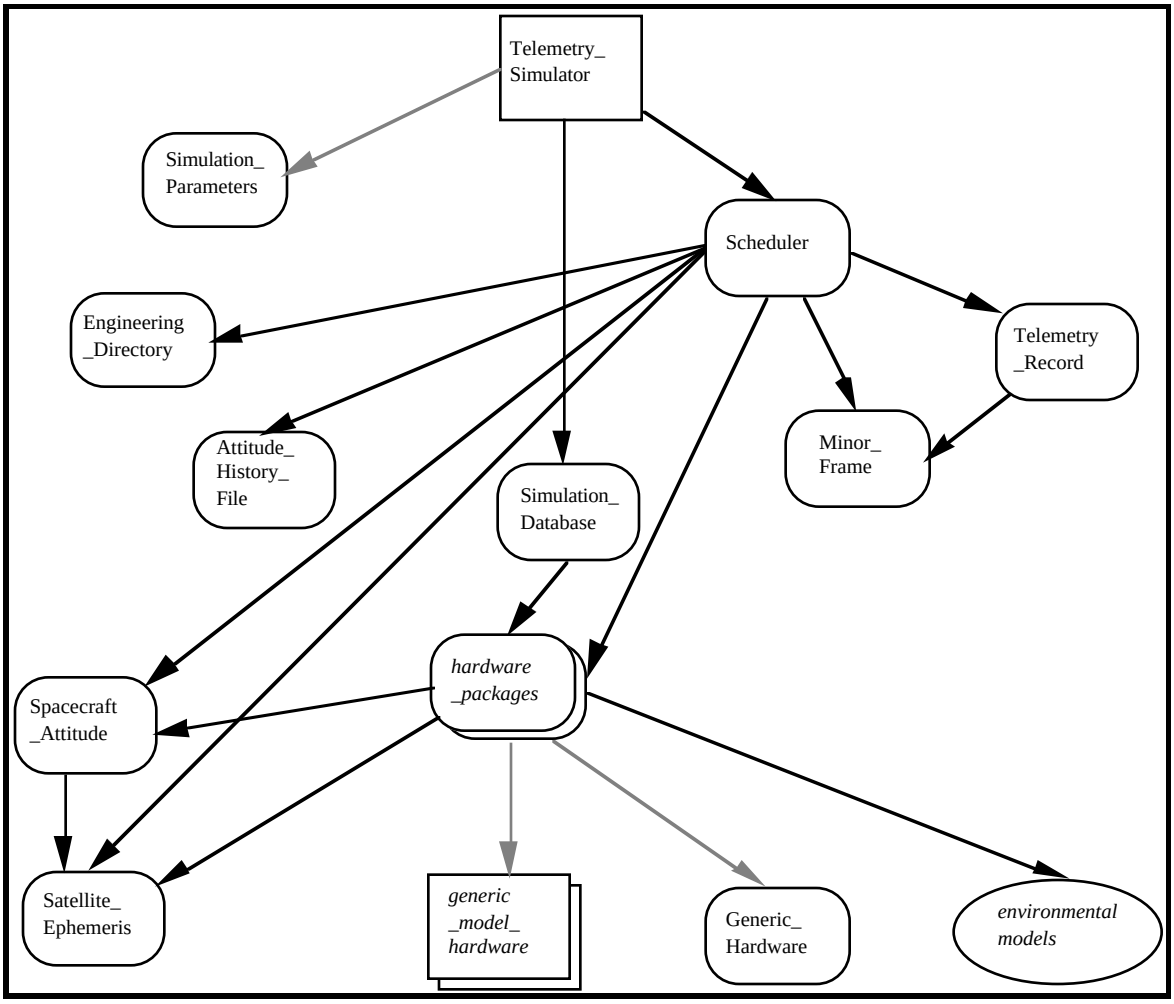


Figure 6. GOOD Object Dependency Diagram

Object Composition Diagrams

An object composition diagram shows the internal structure of a single Ada library unit as well as its external interfaces. In addition to the icons shown in Figure 5, the GOOD notation provides the icons shown in Figure 7. An example of an object composition diagram that uses this notation is shown in Figure 8.

NOTE

The effort required to create and maintain object diagrams — either manually or with a drawing tool — can outweigh their benefit. If a CASE tool that supports object diagrams is unavailable, use textual, object-composition lists from the Ada Compilation System or the VAX Source Code Analyzer instead of object composition diagrams.

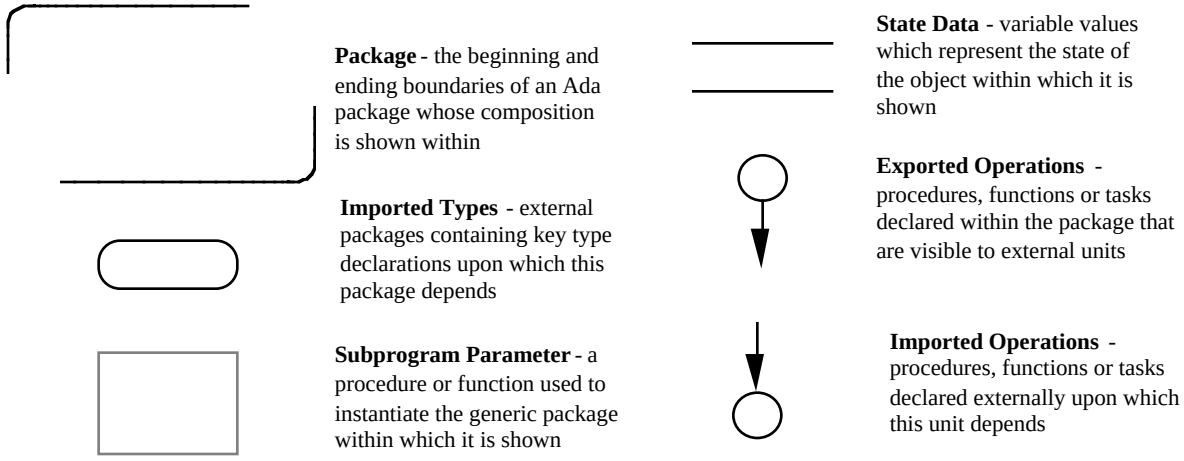


Figure 7. Additional GOOD Graphic Icons for Use in Object Composition Diagrams

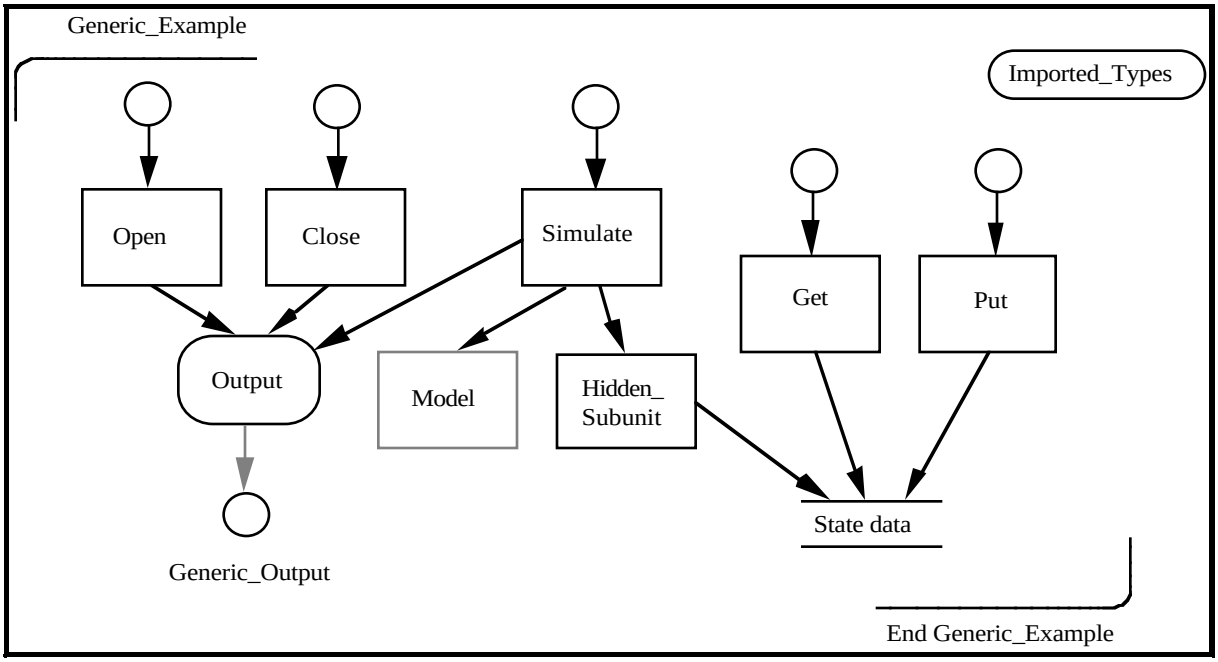


Figure 8. Object Composition Diagram Using the GOOD Notation

NOTE

The inclusion of many object composition diagrams in the design documentation can obscure key aspects of the design. Use object composition diagrams only for those library units that have a complex internal structure, and consider putting them in an appendix.

Task Interaction Diagrams

The purpose of task interaction diagrams is to highlight any concurrency and synchronization in the system design. A task interaction diagram shows the task control hierarchy among all tasks, the main program, and any asynchronous devices. All possible rendezvous are depicted as arrows from the initiating task to the accepting task.

The task interaction diagram uses the same icons as the object dependency diagram shown previously, with the addition of task entry names on the rendezvous dependency arrows. An example of a task interaction diagram is shown in Figure 9. Narrative text should be supplied along with the diagram to describe the interactions among tasks, devices, and the main program.

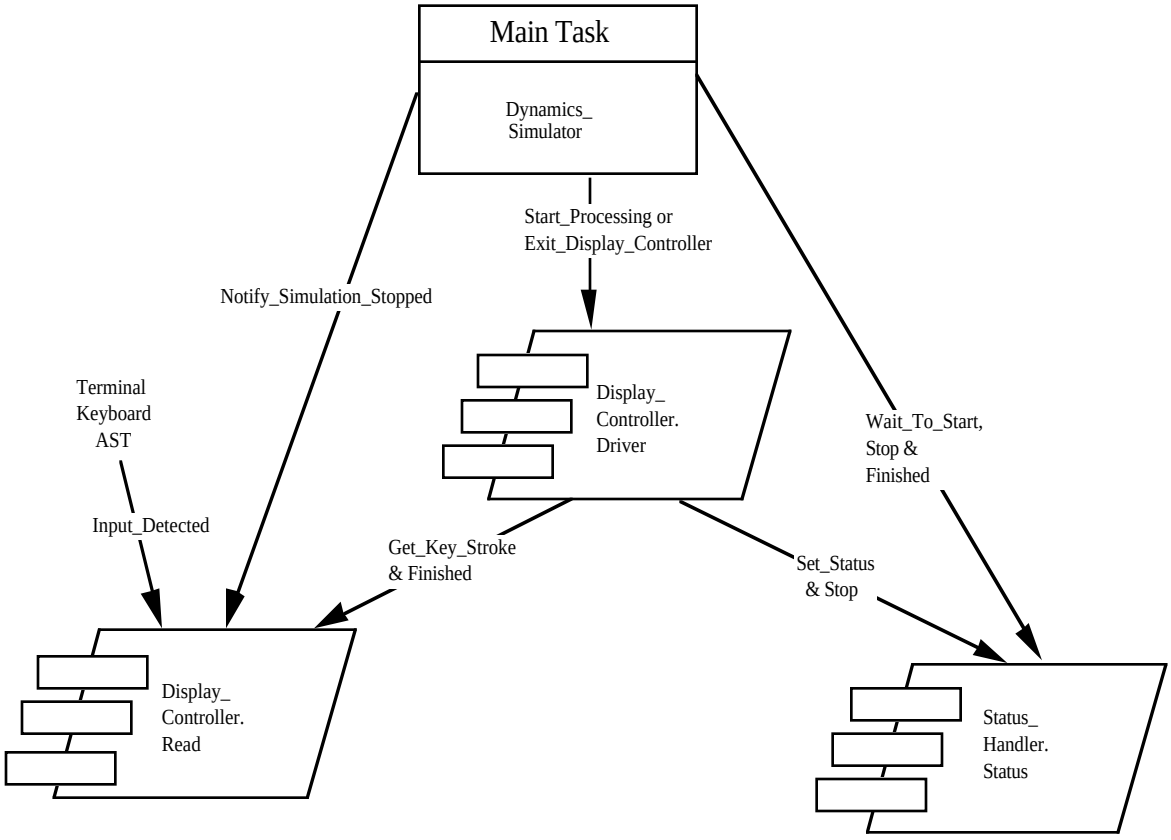


Figure 9. Task Interaction Diagram

Object Prologs

Package and subprogram specifications serve as prologs in Ada systems and are generated for program elements that represent objects in the application domain. The specifications contain comments that provide an object-level description identifying the object and its purpose. The visible data types and operations for the object are declared in Ada code and supplemented by descriptive comments. In addition, the specification contains declarations and descriptive comments for any exceptions that may result from each of its operations or from package initialization code.

All prolog (a.k.a. "header") information is placed at the beginning of the source file before any Ada code (including context clauses, generic parameters, and unit type). The modification history for the component must be included in the prolog/header of the source file, but is not required in the subunits included in that source file.

All prolog/header information relevant to the user also should be included in the specification. Prolog/header information relevant to the implementer or maintainer should be put into the body or subunit as needed.

Table 1 shows the prolog fields for each unit type. Prolog fields are entered in the order shown in the table. For example, the following fields constitute the prolog of a package specification: <*identification*>, *Purpose, Initialization Exceptions, Notes,* and *Modifications.* Information is required for the keywords *Purpose* and *Modifications,* optional for the placeholder <*identification*>, and optional for the keywords *Exceptions* and *Notes.*

Table 1. Prolog/Header Information for Each Type of Ada Unit

	unit type						
prolog field	Package spec	Package body	Procedure or Function spec.	Procedure or Function body	Task spec	Each task entry	Task body
<*identification*>	x	x	x	x	x		x
Purpose *	x		x		x	x	
Initialization Exceptions	x				x		
Exceptions			x			x	
Notes	x	x	x	x	x	x	x
Modifications *	x	x	x	x	x		x
Note: Information must be supplied in fields labeled with an asterisk.							

The *Modifications* field should include the author or maintainer's name, organization, the modification date, and a brief description of the modification. If the software unit is new, the modification field should contain the words "initial version."

Developers can insert information in the prolog (under *Notes*) to assist in tracing the unit back to requirements or specifications. However, references to mission-specific requirements/specifications should *not* be included in generic components.

Developers should use external documentation to facilitate software reuse, rather than providing additional information in the unit prolog. Reuse information can be included in the design document, system description, and user's guide.

Object PDL

During preliminary design, the development team generates high-level PDL for solution-domain objects. The purpose of the PDL produced during preliminary design is to communicate (along with design diagrams) the overall architecture or structure of the solution-domain objects within the system being implemented. As such, the important elements of PDL are data and control dependencies, which are represented by Ada WITH statements, and interface specifications, which are represented by function and procedure declarations for visible operations. Lower-level PDL that describes algorithms and control logic is usually deferred to the detailed design phase.

Ada specifications and PDL are generated using templates and a language-sensitive editor and *may* be compiled to ensure completeness, consistency, and syntactic correctness. Note, however, that compilation of Ada PDL is not mandatory during the preliminary or detailed design phases.

On projects developed in Ada, PDL should include dependency information (i.e., WITHs) as well as any control information (e.g., if, loop, case, calls) that is significant to implementation. Either Ada statements or commentary may be used as PDL. PDL should be generated for the main program, any tasks, and for those units called by either. It should include declaration of types and variables that define an object's state as well as package initialization logic, if any.

Standards for PDL that are appropriate for FORTRAN components may need to be modified for Ada projects. Always use Ada control structures in PDL rather than such non-Ada PDL statements as REPEAT UNTIL.

The identification of units that require PDL in the preliminary design phase may be left to the discretion of the development team. In addition, the development team

may decide to mitigate risk by requiring more detailed PDL for certain units (e.g., drivers, performance-critical units, or units implementing complex algorithms).

Design Walk-throughs and Inspections

Ada projects use design walk-throughs and inspections as described in Sections 5 and 6 of the *Recommended Approach.* Because the design process is highly iterative, the development team holds many informal peer reviews of various design alternatives. Early peer reviews focus on selecting a set of Ada library units that implement the required objects while achieving a good balance among software engineering goals and project drivers. Later peer reviews focus on individual Ada library units and the completeness and consistency of the operations and data types they provide. Once an informal consensus has been reached for a given set of design diagrams or library units, they are formally inspected and certified.

Where reviewers are expected who are new to the object-oriented design paradigm or its notation, the development team should prepare a brief overview of these concepts before their first peer review. This information should be presented to all unfamiliar reviewers before the actual peer review materials are distributed.

The design inspection checklist shown in Figure 6-3 of the *Recommended Approach* should be expanded to include questions intended to isolate any performance problems that may be inherent in the design. For Ada systems, these questions should refer to the efficiency guidelines that appear in the implementation section of this document.

The following additional inspection criteria should be applied to object diagrams:

- Does the object diagram follow the notation and style conventions adopted by the project?

- Are all external references shown?

PRODUCTS

Preliminary Design Report

The primary product of the preliminary design phase of an Ada project is the system architecture, as documented in the preliminary design report. The components of this product are

1. **Object dependency diagrams** that show the system composition and dependencies. These diagrams should include those objects needed to depict the high-level architecture of the system.

2. **Ada specifications** for units depicted in object dependency diagrams.

3. High-level **object data-flow diagrams** (DFDs) to provide the context for major subsystems.

4. **Object composition diagrams** that show the internal structure of Ada library units. Alternatively, the structure of these units may be documented using textual lists such as those produced by the Ada Compilation System (Reference 7) command SHOW PROGRAM or by VAX Source Code Analyzer (Reference 8) command FIND CONTAINED_BY.
5. **Control diagrams** that depict the control hierarchy for the components comprising each major thread of control in the system, and the sequence of component execution. Control diagrams should not introduce objects not depicted on object dependency diagrams, but may show internal components of these objects (i.e., functions or procedures). No standard has been defined for control diagrams, but they are analogous to annotated structure charts, thread diagrams, state transition diagrams, Petri net diagrams, or textual call trees.

6. **PDL**, using Ada code and commentary, for the main procedure, any tasks, and the units that they call. During preliminary design, PDL is usually confined to units that appear on object dependency diagrams

7. **Task interaction diagrams** that show the interfaces between the main procedure and any tasks.

If multiple units share the same architecture (i.e., interface and structure), the development team may elect to produce object diagrams and Ada specifications for only one of the units, as a representative example. Each example object diagram should include a reference to all other, similar units.

Detailed Design Document

The primary product of the detailed design phase is the completed software design, as documented in the detailed design document. The design includes

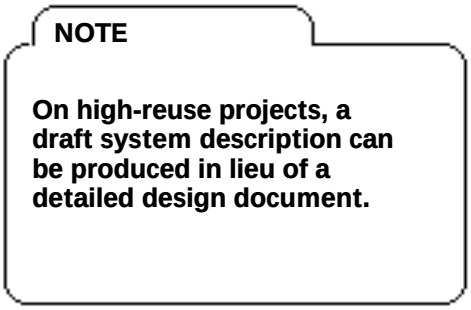
1. Updated **object dependency diagrams**, **object DFDs**, and **control diagrams**

2. Compiled **Ada specifications** for all units

3. **PDL** for all units (Note that compilation of PDL is not mandatory.)

Build Plans

During the detailed design phase, the management team completes the build plan that will be used to implement the design. Section 6 of the *Recommended Approach* describes the contents of the build plan. Because compilation dependencies will affect the order in which components can be implemented, these dependencies must be addressed in the build plan for an Ada project. The plan should also identify the units in each build that will be coded only as stubs.



SUPPLEMENT TO SECTION 7 OF
THE *RECOMMENDED APPROACH*

THE IMPLEMENTATION PHASE

METHODS AND TOOLS

The experience of developers on Ada projects in the flight dynamics environment has resulted in a number of additional guidelines for the implementation phase. These include rules of thumb for coding, guidance for comments within code, and performance guidelines.

Coding Rules of Thumb

- Coding, reviewing, and baselining Ada unit *specifications* from the bottom up will require less recompilation than the same activities will require if they are performed top down.

- Coding and testing Ada unit *bodies* in a top-down fashion — to whatever depth is needed to achieve a given system capability — requires less rework and necessitates development of fewer nondeliverable drivers than coding from the bottom up. Rather than commenting out references within the units being implemented, code the Ada units immediately below the target level as stubs.

- The larger the unit, the harder it is to understand, develop, and maintain. However, better performance optimizations can be achieved by most compilers on a single, large file of Ada units than can be achieved on a collection of separately compiled, smaller Ada units that implement the same capabilities. Weigh other project drivers against the need to improve performance and determine file sizes accordingly.

NOTE

In the flight dynamics environment, all Ada specifications are generally coded and compiled before coding of Ada bodies is begun.

Internal Commentary Guidelines

- Comments should not be redundant with code.
- References should be made to external documents rather than including tutorial or background information as commentary.
- Comments should be used to explain non-obvious code, such as code that has been reorganized for performance reasons.
- Comments should be used to flag deviations from standards or specifications.

Performance Guidelines

The most critical factor for system performance is "having the design and implementation fit the requirements." Design and implementation decisions should always be reevaluated whenever system requirements are modified.

If performance is critical within a particular thread or path in the system, then the development team may have to compromise on engineering goals, such as readability and maintainability, in order to improve performance within the thread. Deviations from good software engineering practice should be kept to an absolute mininum, however, and isolated to the units that need tuning.

To improve performance, developers should focus their attention on

- critical code segments (i.e., those executed many times, such as within loops)
- features that are unique to Ada, such as array slices and exceptions (Unfamiliarity with these features can cause developers to use them in ways that adversely affect performance.)

Developers should avoid the use of exceptions for "non-error" conditions; error conditions are defined to be those with a low probability of occurrence during a "nominal" run. Additionally, developers should review the use of pragma INLINE for

- consistency with compiler vendor's rules for in-line code
- impact on unit dependencies

Code segments that are called frequently (dynamic references) from only a few places (static references) are good candidates for INLINE.

Table 2 is a summary of key efficiency guidelines drawn from four sources:

- *SEL Ada Performance Study Report* (Reference 9)
- *Ada Efficiency Guide* (Reference 4)
- *Ada Style Guide* (Reference 10)
- *VAX Ada Run-Time Reference Manual* (Reference 11)

The rationale for each guideline and a more detailed exposition are provided in the listed reference.

Note that these efficiency guidelines are designed for micro-optimization. That is, they should be applied only to the pieces of the code that need to be optimized, and only when performance is clearly an issue. Also note that some of the guidelines on this list are VAX compiler dependent and may be of limited use in a different environment.

Table 2
Ada Efficiency Guidelines

- Give constrained types preference over unconstrained types. (Reference 9)
- Give static expressions preference over dynamic expressions. (References 9, 10)
- Unroll small loops in frequently used subprograms. (Reference 4)
- Use a *procedure* with an IN OUT parameter in lieu of a *function* that does the same thing. (References 4, 11)
- Change loop-invariant or frequently accessed invariant variables and expressions to constants in locally declared blocks. (Reference 11)
- Use scalar type operations instead of composite type operations. (Reference 11)
- Build values *in place* instead of *within* aggregate expressions. (Reference 11)
- Use RENAMES to remove constraint checks from loops. (Reference 11)
- Use attributes to remove constraint checks from loops. (Reference 11)
- Use records to pass multiple parameters quickly. (Reference 11)
- Use pragma INLINE for small or frequently called subprograms. (References 4, 9, 10, 11)
- Use generic value parameters to allow reusers to choose between accuracy and efficiency. (References 4, 9, 11)
- Process multidimensional arrays in row-major order. (Reference 9)
- Use short-circuit control forms or nested IF statements when boolean expressions contain function calls that have no side effects. (Reference 9)

SUPPLEMENT TO SECTIONS 8 AND 9 OF
THE *RECOMMENDED APPROACH*

THE SYSTEM AND ACCEPTANCE TESTING PHASES

Even on Ada systems with high reuse, the test team must verify the correct execution of the entire system, not just the new or modified components of the system. Consequently, testing an Ada system is substantially the same as testing a system developed in FORTRAN. The same methods and tools described in Section 8 of the *Recommended Approach* apply.

PRODUCTS

Software Reuser's Guide

A *software reuser's guide* is produced during the system testing phase to accompany an Ada library unit that is to be placed in a reuseable software library (RSL). Because of the effort required to create and maintain understandable and reliable documentation, the development team should create a software reuser's guide only for reusable software with a low probability of change or a high frequency of reuse.

The software reuser's guide is organized by library unit. For each library unit, it includes

- an overview of the library unit;
- a description of each element (e.g., subprogram, generic parameters, types);
- a copy of the library unit's specification; and
- examples of its use.

NOTE

The task of creating a reuser's guide is normally allocated to the team that is responsible for creating and maintaining the RSL.

The overview presents the purpose of the library unit, describes key concepts, and highlights aspects critical to understanding the unit. This section also describes interactions, dependencies, and limitations significant to the unit's use. Where appropriate, this section could include graphical representations and refer to background material.

Each element of the library unit is described in a separate subsection. The elements are presented in a logical order, such as instantiation, elaboration exceptions, and nested subunits, but the order may differ from that in the specification. The subsection for each element describes what the element does, how to use it, and any exceptions that might result. Private or hidden parts not significant to a user's understanding need not be described. Usage examples can be included along with the description of library unit elements or following the copy of the library unit's specification. For an example of a description of a reusable library unit, see the description of package TEXT_IO in the *Reference Manual for the Ada Programming Language* (Reference 12).

ACRONYMS

APWG	Ada Process Working Group
CASE	computer-aided software engineering
CDR	critical design review
CMS	Code Management System
CSC	Computer Sciences Corporation
DEC	Digital Equipment Corporation
DFD	data flow diagram
GOOD	general object-oriented development
GSFC	Goddard Space Flight Center
I/O	input/output
LSE	language-sensitive editor
NASA	National Aeronautics and Space Administration
OOD	object-oriented design
PDL	program design language (pseudocode)
PDR	preliminary design review
RSL	reusable software library
SDMP	software development/management plan
SEL	Software Engineering Laboratory
TBD	to be determined

REFERENCES

l. Software Engineering Laboratory, SEL-81-305, *Recommended Approach to Software Development (Revision 3),* L. Landis, S. Waligora, F. McGarry, et al., June 1992

2. G. Booch, *Software Engineering with Ada (Second Edition),* Benjamin/Cummings: Redwood City, CA, 1987

3 P. Coad and E. Yourdon, *Object-Oriented Analysis,* Yourdon Press: NY, 1991

4 Flight Dynamics Division Code 550, 552-FDD-91/068R0UD0, *Ada Efficiency Guide,* E. Booth, August 1992

5. G. Booch, *Object-Oriented Design (with Applications),* Benjamin/Cummings: Redwood City, CA, 1991

6. Software Engineering Laboratory, SEL-86-002, *General Object-Oriented Software Development,* E. Seidewitz and M. Stark, August 1986

7. *Alsys 286/386 UNIX Ada Compilation System, Users Guide (Version 4),* Alsys, Inc., Burlington, MA, 1989

8. Digital Equipment Corporation, AA-DB34J-TE, *VAX Language Sensitive Editor and VAX Source Code Analyzer User Manual,* Maynard, MA, September 1990

9. Software Engineering Laboratory, SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report,* E. Booth and M. Stark, July 1991

10. —, SEL-87-002, *Ada Style Guide (Version 1.1),* E. Seidewitz et al., May 1987

11. Digital Equipment Corporation, AA-EF88B-TE, *VAX Ada Run-Time Reference Manual,* Maynard, MA, May 1989

12. *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD 1815A, February 1983